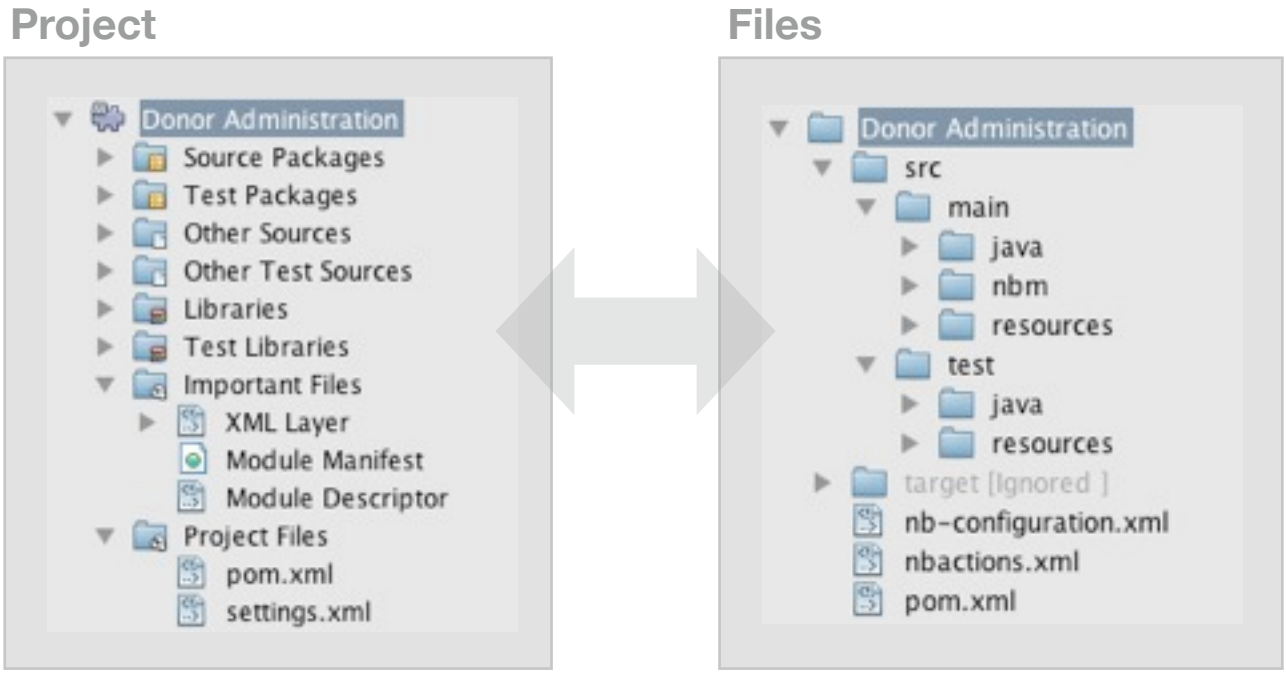
INTRODUKTION TILL
PROSANGUTVECKLING

ProSang
BLODCENTRALSSYSTEM

Johan Andrén 2010-05-17

INNEHÅLL

PROJEKTSTRUKTUR



Filstrukturen i mavenprojekt ser i princip alltid lika ut. I NetBeans visas en projektvy där olika typer av kataloger från våra projekt grupperas ihop. NetBeans har också en filvy där man kan se hur katalogerna och filerna faktiskt är ordnade på hårddisken.

Source Packages innehåller själva javaklasserna. Här finns aldrig något annat än javaklasser

Test Packages innehåller javaklasser med unittester för klasserna från Source Packages

Other Sources innehåller resurser. T.ex. textfiler, xml-filer, .properties-filer, bilder.

Other Test Sources resurser som bara behövs för unittesterna.

Libraries är ingen riktig katalog utan visar projektets externa beroenden. De faktiska beroendeposterna är listade med <dependency>-block i projektets pom.xml

Test Libraries samma sak men innehåller de bibliotek som bara behövs när testerna körs.

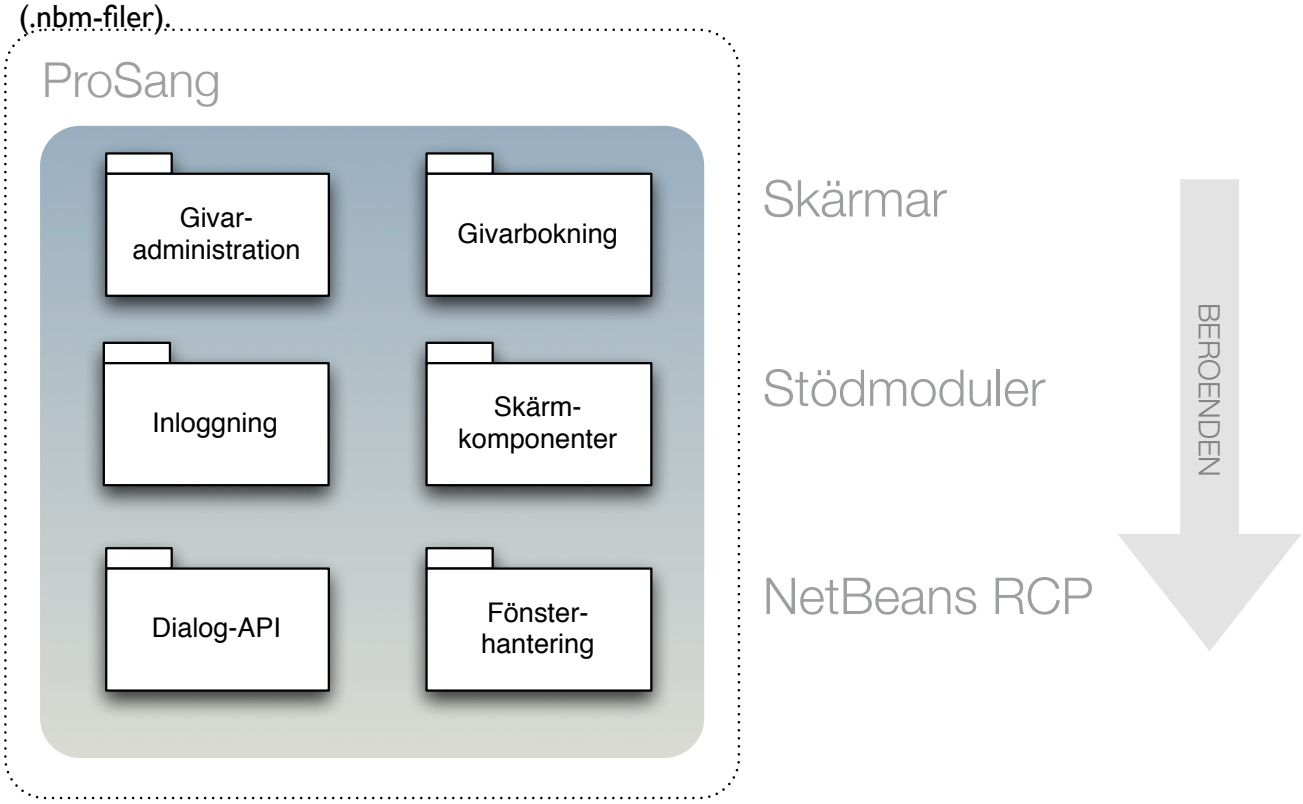
Important Files - NetBeans RCP-specifika konfigurationsfiler. layer.xml som låter oss publicera funktioner i våra moduler så att de kan användas av NetBeans RCP.

Project Files - Mavens konfigurationsfiler, pom.xml är från projektet men settings.xml är en genväg till din lokala inställningsfil som egentligen ligger i $HOME/.m2/settings.xml

NETBEANSRCP

ProSang-klienten är byggd ovanpå NetBeans RCP (Rich Client Platform). NetBeans RCP är en opensourceplatform för att bygga grafiska Javaapplikationer som ägs och utvecklas av Sun/Oracle. Utvecklingsmiljön NetBeans IDE bygger precis som ProSang på NetBeans RCP.

En applikation som skrivs för NetBeans RCP är uppbygd av smådelar som kallas moduler (.nbm-filer).



I ProSang är modulerna uppdelade i olika kluster.Vi har ett kluster som heter Core som innehåller moduler med komponenter och API:er som är gemensamma för hela ProSangklienten.

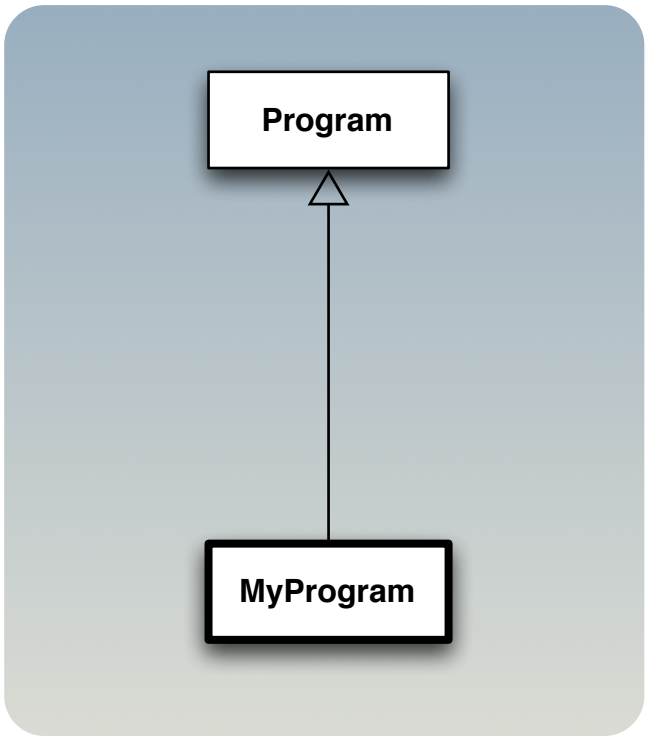
För varje rutin (Administration, Givare, Patient etc.) finns sedan ett eget kluster. Modulerna som innehåller själva skärmarna skall inte dela med sig någon funktionalitet till andra moduler Gemensam funktionalitet som bara berör modulerna i ett kluster placeras i en modul som heter common i klustret. Funktionalitet som är generell för hela klienten placeras i olika moduler under core-klustret. Exempel på det är ProSangs sök-API, resultat-API.

Moduler har ett rikare stöd för att tala om vilka klasser som är till för omvärlden än vanliga jar-filer. Man bör bara dela ut de paket som innehåller genomtänkta publika API:er.Vilka paket som är publika ställer man in i modulens pom.xml

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>nbm-maven-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <publicPackages>
            <!-- All other packages are private to the module -->
            <item>se.databyran.prosang.mymodule.package.**</item>
        </publicPackages>
```

PROGRAMAPI

ProSang består av en mängd delprogram som tillhandahåller olika funktioner till olika personer på blodcentralen. Sådana delprogram kallas för åtgärdskoder efter den kod de startas med. En sådan kod kan se ut såhär t.ex. ”G100”. G står för Givarrutin och 100 är en unik identifierare för vilket givarprogram det handlar om.



Varje åtgärdskod i ProSang är implementerad med en subklass till Program som talar om vilken kod, vilken ikon och vad programmet heter. Klassen har också en metod som anropas när åtgärdskoden startas och som öppnar själva skärmen för åtgärdskoden.

För att ProSang skall hitta programmet och visa det i programträdet måste programklassen registeras som implementation av SPI:t (Service Provider Interface) i en fil under META-INF/services som heter samma sak som själva SPI-interfacet den implementerar.

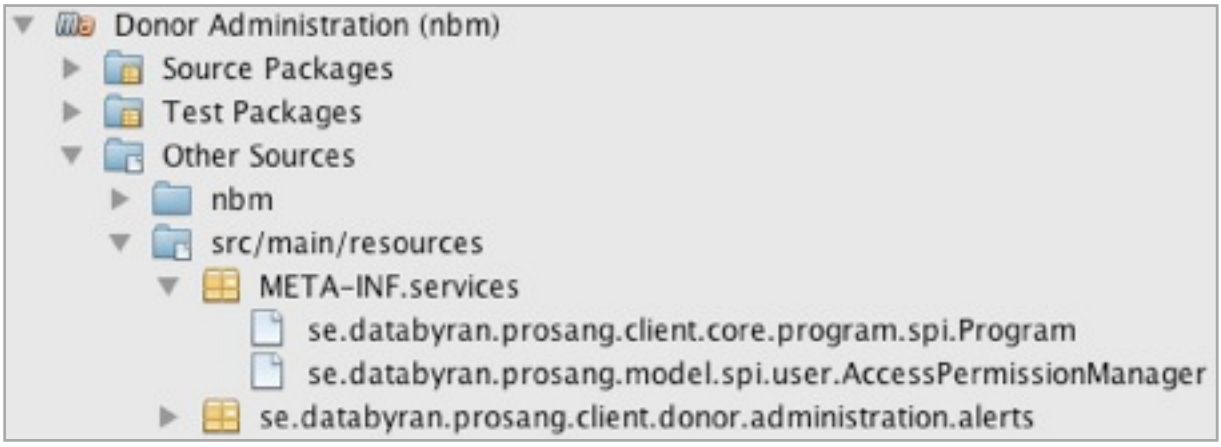
För ett program heter filen såhär:

```
META-INF/services/se.databyran.prosang.client.core.program.spi.Program
```

Registreringsraden för ett program i filen kan se ut såhär:

```
se.databyran.prosang.client.addressbook.AddressBookProgram
```

Samma modul kan innehålla flera program, de listas i samma fil med en ny rad för varje program. I NetBeans hittar du registreringsfilen under ‘Other Sources’:



Programmet
Hello World!

Säg att vi höll på att skriva programmet ”G001 Adressboksprogrammet”, då skulle vår programklass kunna se ut såhär:

```
public class AddressBookProgram extends Program {

    public ProgramId getProgramId() {
        return new ProgramId(ProgramCategory.DONOR, "001");
    }

    public String getLabel() {
        return "Addressboksprogrammet";
    }

    protected IconType getIcon() {
        return IconType.ADDRESS_SMALL; // hittepå
    }

    public String getDescription() {
        return "Hantera addresser till vänner och bekanta";
    }

    protected void startProgram() {
        System.out.println("Hello world!");
    }
}
```

I18N är en förkortning för internationalization som innehåller 18 bokstäver mellan I och N.

.properties-filerna syntax kommer från javas klassbibliotek och kan läsas och skrivas med klassen Properties.

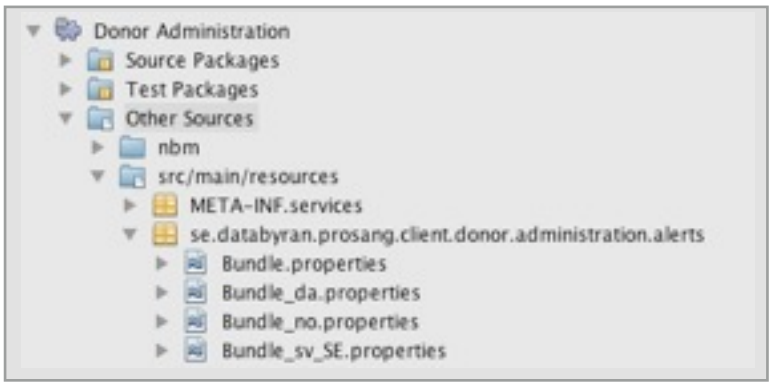
Locale är en kombination av land och språk (vissa länder har flera språk) som t.ex. kan se ut så här för sverige ”sv_SE”

I18N

Alla strängar som visas för användaren i ProSang måste internationaliseras. Java har inbyggt stöd för internationalisering med Bundle-API:t men vi använder ett bibliotek ovanpå det som kommer från NetBeans RCP. Själva texterna ligger i enkla textfiler med nyckel och text till varje sträng. Varje språk vi har stöd för finns i en separat sådan fil för varje java-paket i ProSang.

Filnamnet talar om vilket språk som finns i den, med undantag för den propertiesfil som inte har något språk pålagd och som är standardspråket som java faller tillbaka på att använda ifall en nyckel saknas i språket applikationen körs på. Alla våra språkfiler skall ha ett namn som börja med ”Bundle”.

Bundle.properties innehåller engelska, vilket kan tyckas lite märkligt jämfört med gamla prosang men tanken är att det skall underlätta den dagen vi behöver översätta till ett språk utanför norden.



Java använder den locale som systemet är inställt för men man kan också trumfa det med en flagga som man skickar med när man startar en javaapplikation för att köra ett program på norska även om man kör datorn på svenska. Localen används också för att t.ex. formatera datum och tid på rätt sätt.

```
Bundle.properties:
MyTopComponent.title=My Top Component
MyTopComponent.countField.text=There are {0} items in the list

Bundle_sv_SE.properties:
MyTopComponent.title=Min toppkomponent
MyTopComponent.countField.text=Det finns {0} saker i listan
```

För att läsa filen använder man sedan NbBundle.getMessage(Class class, String keyName)

```
NbBundle.getMessage(MyTopComponent.class, ”MyTopComponent.title”); // NOI18N
NbBundle.getMessage(MyTopComponent.class,
                    ”MyTopComponent.countField.text”, // NOI18N
                    5);
```

Strängar som ligger i källkoden men som inte skall översättas skall markeras med // NOI18N på så sätt kan vi hitta strängar som vi missat att översätta. Bundlenycklarna skall börja på klassen de används i och sedan vara någon form av objektsökväg till fältet som innehåller texten.

Loggning

Ibland vill man skriva ut lite spårutskrift från sitt program. I java finns System.out.println som skriver en rad till standard out. Den skall man aldrig använda, istället skall man använda Log4J som är det log-paket vi använder för utskrift.

Fördelen med att använda ett logpaket för utskrifter är att de vet var i koden logmeddelandet kom ifrån och att de går att styra och konfigurera på olika sätt.

Lognivå	Beskrivning
ERROR	Fel som inte går att kringå på något sätt och som leder till att programmet kraschar eller blir funktionsodugligt
WARN	Fel som programmet hanterar på något sätt men som borde åtgärdas
INFO	T.ex. hur en komponent är konfigurerad, när applikationen startas
DEBUG	Spårutskrifter som inte ger så mycket data.
TRACE	Spårutskrifter som inte ger massvis med data. T.ex. från insidan av en for-loop.

Loggningen knyts till vilken klass den kommer från när man skapar sin logger. I en konfigurationsfil styr man sedan vilka loggers meddelanden som skall skrivas till vilken logfil och vilka nivåer som skall filtreras bort.

I ProSang-projektet finns en Log4J-konfiguration för utveckling som ligger i applikationsprojektet. Man kan också ansluta till en JMX-böna i ProSang-klienten för att ställa lognivåer i runtime. På serversidan finns en konfigurationsfil per serverinstans där man kan ställa lognivåer. Även där kan man ansluta till en JMX-böna för att ställa lognivåer i runtime.

Java innehåller ett eget log-API som är lite osmidigare att använda än Log4j. Dessutom är Log4j integrerat med JBoss som är vår applikationsserver.

```
public class MyClass {

   private static final Logger LOGGER =
                            Logger.getLogger(MyClass.class);

   public void myMethod() {
      LOGGER.info("Entered my method"); // NOI18N

      try {
          someOtherMethod();
      } catch (OtherMethodException ex) {
          LOGGER.warn("Something failed", ex); // NOI18N
      }

      ..
```

Programmet
Hello World! version 2

```
public class AddressBookProgram extends Program {

    private static final Logger LOGGER =
                        Logger.getLogger(AddressBookProgram.class);

    public ProgramId getProgramId() {
        return new ProgramId(ProgramCategory.DONOR, "001"); // NOI18N
    }

    public String getLabel() {
        return NbBundle.getMessage(AddressBookProgram.class,
                                   "AddressBookProgram.label"); // NOI18N
    }

    protected IconType getIcon() {
        return IconType.ADDRESS_SMALL;
    }

    public String getDescription() {
        return NbBundle.getMessage(AddressBookProgram.class,
                               "AddressBookProgram.description"); // NOI18N
    }

    protected void startProgram() {
        LOGGER.info(”Hello World!!!”); // NOI18N
    }
}
```

TESTDRIVEN

Att arbeta testdrivet innebär i princip att man aldrig skriver någon kod som inte är till för att få ett test att gå igenom. Genom att börja med ett ett grundläggande test och sedan skriva kod som får testet att gå igenom och sedan utöka testet med komplexare och komplexare krav på funktionalitet så hjälper man sig själv och andra på ett antal sätt:

- Det blir ett kvitto på att det man kodat faktiskt fungerar, som till skillnad från ett manuellt test går att återupprepa kontinuerligt och dessutom oerhört mycket snabbare varje gång
- Det blir ett skyddsnät för dig själv och andra som visar att ny funktionalitet inte föstör något som fungerade förut
- Det är en form av dokumentation som visar hur ditt publika API är tänkt att användas
- Det hjälper dig att tänka som en konsument av den tjänst du håller på att implementera, är mitt publika API verkligen vettigt att använda?
- Testbar kod är ofta kod som är bättre designad än kod som inte går att testa.

Därför är det extremt viktigt att du hela tiden försöker arbeta testdrivet.

En testmetod skall vara isolerad och inte påverkas av andra tester och inte heller påverka andra tester. Testet skall namnges så att det går att utläsa som den funktion det testar.

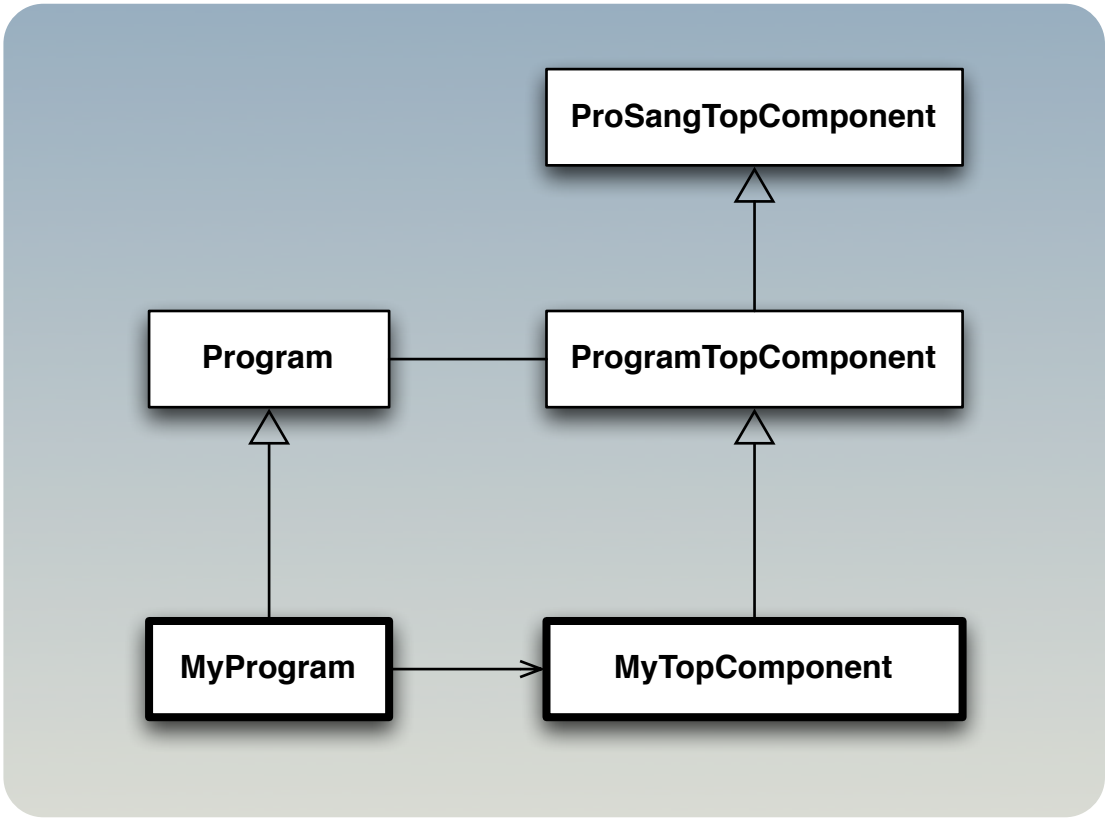
```
...
import static org.junit.Assert.*;

public class MyClassTest {

   @Test
   public void beforeAnySearchHasBeenPerformedTheResultIsNull() {

      Searcher instance = new Searcher();
      String result = instance.getResult();

      assertNotNull(result);
   }

   ...
```

För att unittesta våra klasser använder vi JUnit 3 och 4 som är väl integrerade med NetBeans IDE och Maven.

Testerna körs varje gång projektet byggs och på så sätt kan vi vara säkra på att vi upptäcker fel som vi skapar i kod som förut fungerat utan att manuellt validera funktioner varje gång vi ändrar något.

Testerna körs också på vår ci-server http://smith.databyran.local:9000/hudson efter varje incheckning till källkodsrepositoriet.

SKAPA EN SKÄRM



Nästan varje program i ProSang har en skärm. Varje sådan skärm ärver ProgramTopComponent.

NetBeans guide för att skapa topkomponenter lägger till en massa text i olika konfigurationsfiler som vi inte vill ha och vi har ingen egen mall för topkomponenter. Istället skapar man en ny JPanel (som man kallar MittProgramNamnTopComponent) och ändrar så att den ärver ProgramTopComponent i java-koden.

Begreppet TopComponent kommer från NetBeans RCP som har en egen fönsterhanterare som sköter de interna fönstren i en NetBeans RCP-applikation.

Programmet
Skapa och starta skärmen

```
public class AddressBookProgram extends Program {

    public ProgramId getProgramId() {
        return new ProgramId(ProgramCategory.DONOR, "001");;
    }

    public String getLabel() {
        return NbBundle.getMessage(AddressBookProgram.class,
                                   "AddressBookProgram.label");
    }

    protected IconType getIcon() {
        return IconType.ADDRESS_SMALL;
    }

    public String getDescription() {
        return NbBundle.getMessage(AddressBookProgram.class,
                                   "AddressBookProgram.description");
    }

    protected void startProgram() {
        ProgramTopComponent topComponent = new AddressBookTopComponent();
        run(topComponent);
    }
}
```

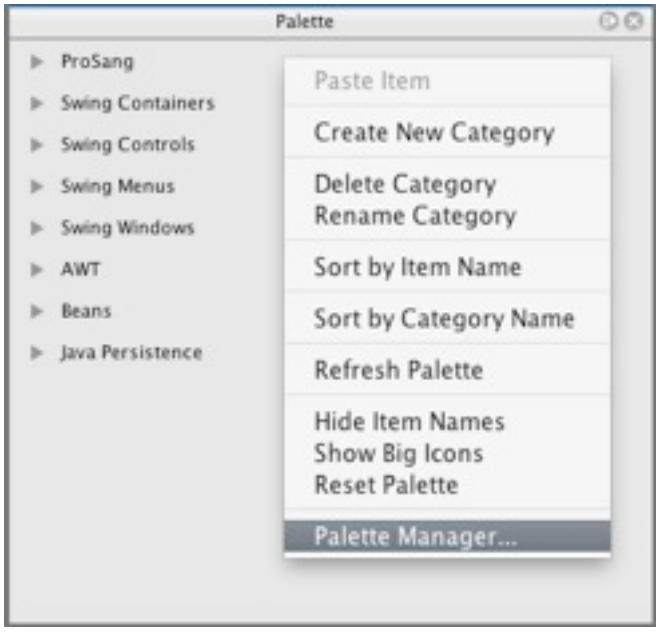
Topkomponenten

```
public class AddressBookTopComponent extends ProgramTopComponent {

    ...

    public AddressBookTopComponent() {
        initComponents();
        setModel(presentationModel);
        ...
    }

    ...
}
```

KOMPONENTER

ProSangs specialkomponenter

Komponent	Beskrivning
ProSangSimpleList	En tabell med data för visning, enkelt API för knappar som utför operationer på listan
ProSangTextField	Textfält med funktioner för formattering, validering och notifiering
ProSangDatePickerField	Datumväljare med kalender
DefinitionSelector	Autokompletekombobox för definitioner
EnumSelector	Autokompletekombobox för enums
ProSangDualList	Två listor, en med möjligt urval och en med de som är valda.



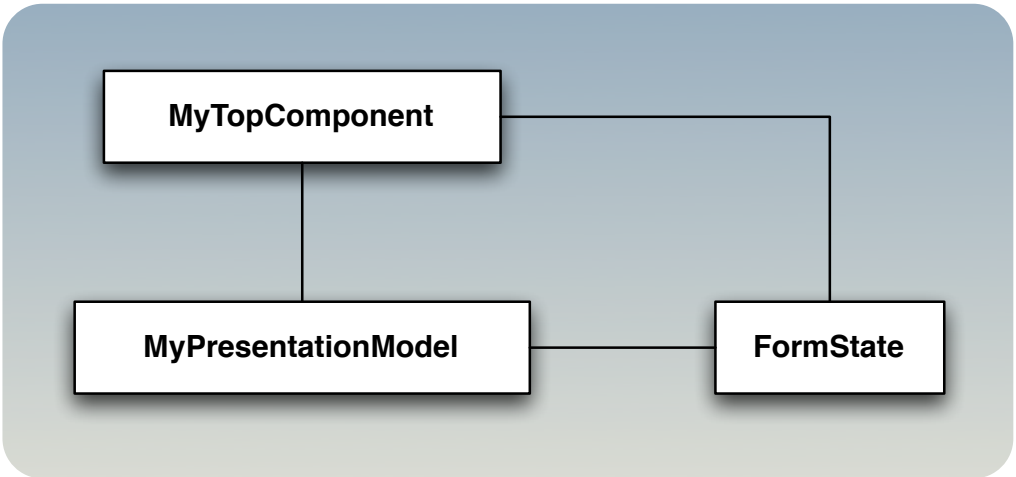
De flesta komponenterna i ProSang kommer från modulerna Client Modules/Core Utilities och Client Modules/Core Components. För att kunna rita skärmar med ProSangs specialkomponenter måste du själv lägga till dem genom att i skärmritaren högerklicka på paletten och välja ”Palette Manager”.

Skapa sedan en kategori som heter ProSang som du kan placera alla ProSang-komponenter i och lätt hitta dem.

Klicka sedan på ”Add from jar...”, leta upp jarfilen för ”Core Utilities” på din hårddisk. När du valt jarfilen kommer du få upp en jättelång lista med klassnamn. NetBeans kan inte skilja på vilka klasser som är skärmkomponenter och vilka som är vanliga klasser så du får leta fram komponenterna du vill lägga till. Listan ovan är en bra startpunkt.

Gör samma sak med Core Components.

MEROMSKÄRMAR



PresentationModel

Alla skärmar med data och knappar kan sägas ha ett tillstånd. Tillståndet kan vara en lista med objekt som syns i skärmen eller om en knapp går att klicka på. Det är också operationer som knappar utför.

För att underlätta testning och spaghetti-kod i skärmklasserna representerar vi alltid skärmens innehåll och tillstånd med en subklass till StandardPresentationModel.

FormState

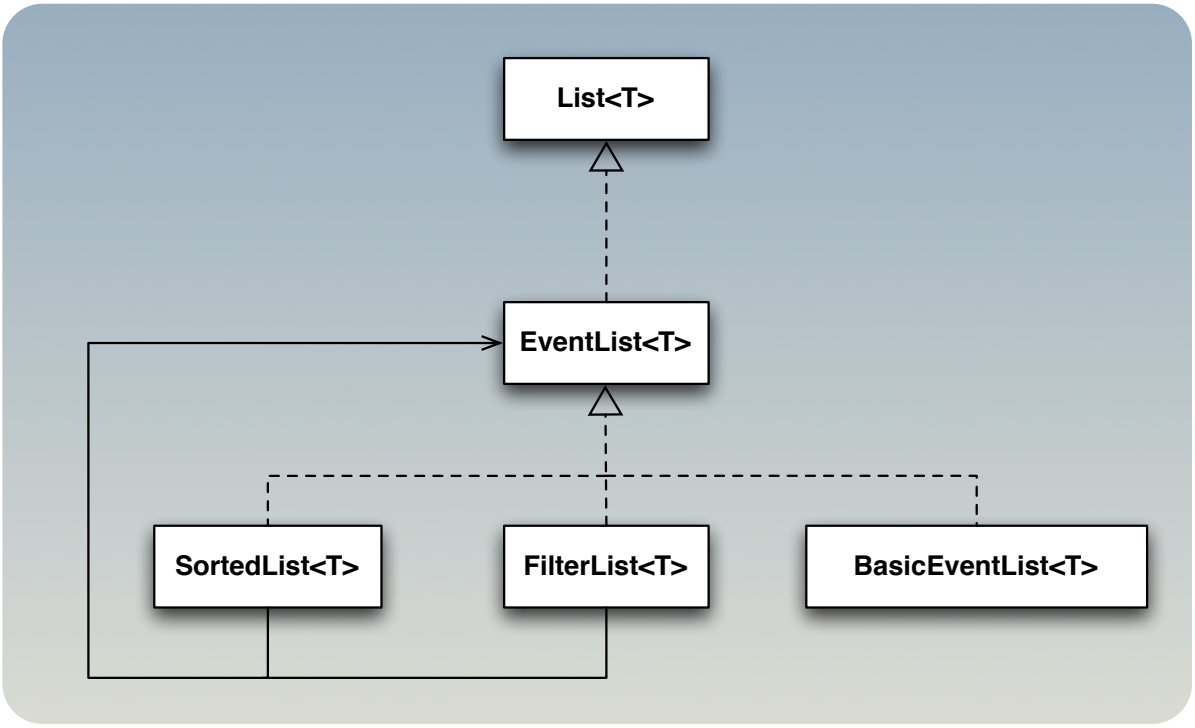
Alla skärmar med data som går att förändra på något sätt har ett skärmtillstånd. Vi har gjort en representation av detta med klassen FormState.

Ett formstate kan vara oförändrat, ändrat, innehålla fel eller både vara ändrat och innehålla fel. Ofta skall t.ex. en ok-knapp vara omöjlig att klicka på om inget ändrats eller om något ändrats men skärmen innehåller ett fel.

FormState gör det möjligt för våra skärmar att kontinuerligt veta om de innehåller fel och förhindra att man ens kan klicka på t.ex. Ok-knappen så länge skärmen inte innehåller ändringar eller innehåller fel.

```
public class AddressBookTopComponent extends ProgramTopComponent {


    public AddressBookTopComponent() {
        initComponents();

        FormState formState = new MockFormState();
        setFormState(formState);
    }
```

Listor
Glazed Lists



I många skärmar vill vi ha filtrering av tabeller och möjliga val i komboboxar. Vi vill också kunna förändra listors innehåll så att listor som finns i skärmen uppdateras.

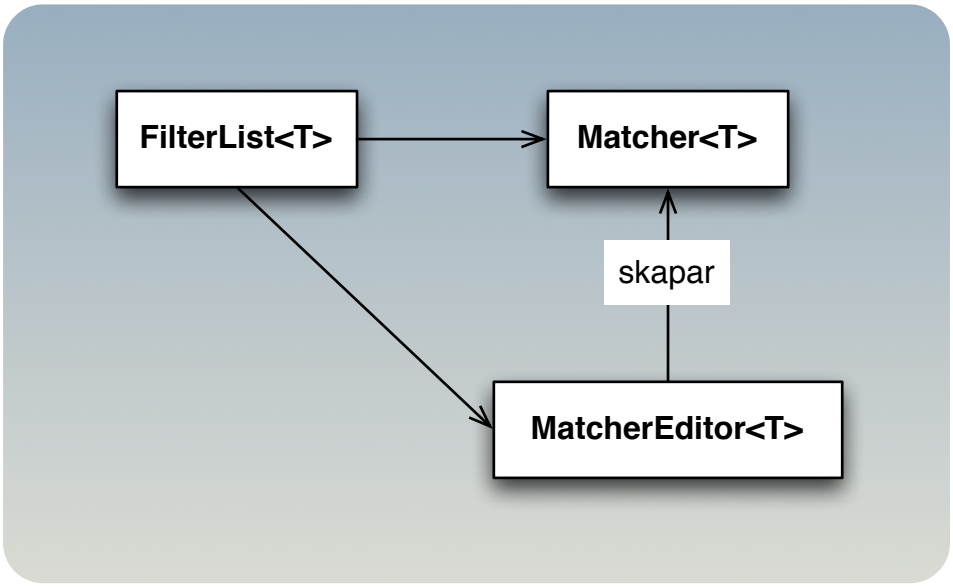
De listor som finns i javas klassbibliotek saknar stöd för det.Vi använder därför ett bibliotek som heter Glazed Lists som utökar javas Collection-API med listor som avfyrar event när objekt i listan läggs till, flyttas eller tas bort (med lite special även om egenskaper på objekt i listan ändras)

Våra tabeller är helt byggda kring glazed lists listor.

Eventlistpaketet använder trådar bakom kulisserna så för att göra dem trådsäkra behöver du låsa dem innan du förändrar dem, annars kan du få konstiga trådfel eller ConcurrentModificationException.

```
EventList<Person> addressBook = new BasicEventList();


...

public void clearAddressBook() {
   addressBook.getReadWriteLock().writeLock().lock();
   addressBook.clear();
   addressBook.getReadWriteLock().writeLock().unlock();
}

...
```

Filtrering




```
EventList<Person> addressBook = new BasicEventList();
EventList onlyAPersons = new FilterList(list, new Matcher<Person>() {

    public boolean matches(Person person) {
        return person.getName().startsWith("A");
    }
})
```

Om man har en statisk filtrering (där det man filtrerar med inte förändras) sätter man en matcher på sin filterlista.

När man vill ha ett formulär som filtrerar om listan gör man det med en MatcherEditor. MatcherEditorn skapar en ny Matcher och informerar listan om att sökvilkoren förändrats så att listan filtreras om. Man kan även sätta en ny Matcher på filterlistan direkt om man har tillgång till den.

```
private FilterList<Person> addressBook = ...


private void setSearchText(String searchText) {
   addressBook.setMatcher(new NameMatcher(searchText));
}
```

TableFormat
Hur skall tabellen se ut?

För att styra vilka kolumner som skall finnas i en tabell skapar man en ColumnsTableFormat och i den stoppar man en TableColumn för varje kolumn som tabellen skall ha.

```
...

private ProSangTableFormat createTableFormat() {
   TableColumn nameColumn = new TableColumn<Address, String>(
       ”Namn”,
       50,
       String.class) {

       public String getValue(Address address) {
             return address.getName();
        }
   }


   TableColumn streetColumn = new TableColumn<Address, String>(
       ”Gata”,
       50,
       String.class) {

       public String getValue(Address address) {
             return address.getStreet();
        }
   }

   return new ColumnsTableFormat<Address>(new TableColumn[] {
                                                 nameColumn, streetColumn});
}

...
```

Kolumnens innehåll

Tabellens innehåll

Många tabellkolumner återkommer i flera tabeller, sådana kolumner för t.ex. givare kan du hitta i modulerna Donor Common eller Core Tables. Om du skapar en kolumn som du kan tänka dig kommer behövas på fler ställen, tänk på att stoppa den i någon av de moduerna!

ProSangSimpleList
Koppla ihop listan med skärmen

För att visa en lista i skärmen använder vi vår egen komponent ProSangSimpleList. Du sätter upp innehållet i din skärm i konstruktorn. Eftersom listan avfyrar event när innehållet förändras skall du göra detta även om din lista får data först senare.

```
public class AddressBookTopComponent extends ProgramTopComponent {

   public AddressBookTopComponent() {
       initComponents();

       addressSimpleList.setup(model.getAddresses(), createTableFormat());

       ...
   }

...
```

ProSangSimpleList är förberedd med de vanligaste knapparna (lägg till, ta bort och redigera). Så för att skapa en sådan lista registrerar man bara olika callbacks för lägg till och ändra. Ta bort kan den sköta helt själv men du kan behöva skapa en ObjectRenderer för att göra om raden som tas bort till en användarvänlig text (”Vill du verkligen ta bort .....”).

Om man har mer specifika knappbehov så kan man implementera interfacet ButtonDescription och skicka in till sin ProSangSimpleList.

```
public class AddressBookTopComponent extends ProgramTopComponent {

   public AddressBookTopComponent() {
       initComponents();


       addressSimpleList.setItemCreator(new ItemCreator() {
            public Object create() {
               // TODO show a dialog
            }
       }
       addressSimpleList.setItemEditor(new ItemEditor() {

            @Override
            public Object edit(Object original) {
               // TODO copy object, show dialog
            }
        });
   }

...
```

DIALOGAPI

Dialogfabriken

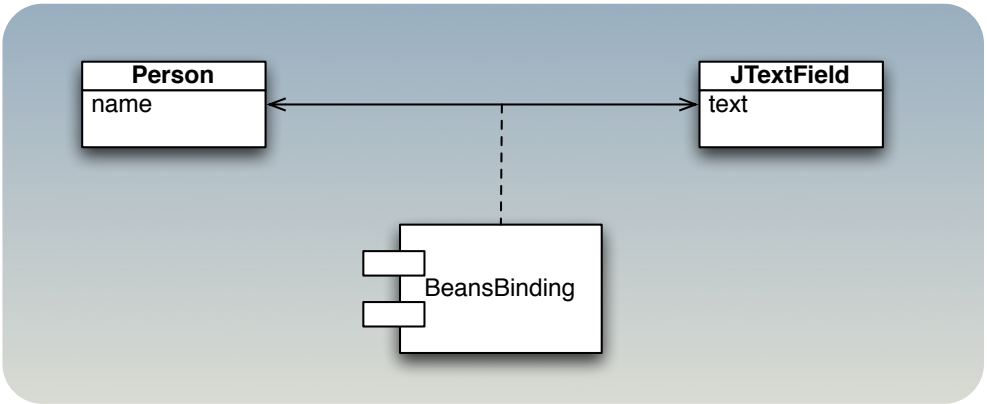
DialogFactory är ett lite olyckligt namn eftersom det inte alls är någon factory utan snarare en statisk utilityklass. Bakom kulisserna anropar den NetBeans platforms dialog API.

För att alla dialoger i ProSang skall se likadana ut så skall alla knappar visas med hjälp av klassen DialogFactory. För enkla dialoger som sådana med en fråga och ett par knappar finns särskilda metoder. I de fall en mer komplicerad dialog behöver visas kan man skicka in en egen panel till DialogFactory, panelen innehåller då inte knapparna som skall finnas i nederkanten av dialogen utan dem skickar man med som ett separat argument till DialogFactory.

Precis som skärmar har dialoger med egna paneler ett skärmtillstånd (FormState) som t.ex avgör om man kan klicka på ok eller inte. För att automatiskt knyta ihop en panel med dialogknapparna

```
// simple dialogs
DialogResult result = DialogFactory.yesNoCancelDialog(”titel”, ”fråga”);
if (result == DialogResult.YES) {
   // save
)


boolean yes = DialogFactory.yesNoDialog(”titel”, ”fråga”);


boolean ok = DialogFactory.okCancelDialog(”titel”, ”påstående”);


DialogFactory.warningDialog(”varning”);


// dialog with custom panel
MyCustomPanel panel = new MyCustomPanel();
if (DialogFactory.okCancelDialog(”titel”, panel)) {
   Address value = panel.getAddress();
   // save value
}
```

BeansBinding
Synkronisera bönor



För att knyta ihop presentationsmodellen med skärmen använder vi ett bindningsframework som heter BeansBinding. Beansbinding använder javas PropertyChange-API för att upptäcka att ett fält på ett objekt ändrats och sedan synkronisera värdet med ett fält i ett annat objekt.

För att ändringar av värdet på ett fält skall upptäckas måste det uppfylla några krav som kommer från en gammal java-komponent-spec som heter JavaBeans. Ett fält som uppfyller dessa krav kallas ibland för en property - då avses både själva fältet, gettern och settern.

```
public class MyClass {

   private final PropertyChangeSupport propertyChangeSupport
                        = new PropertyChangeSupport(this);

   public static final String PROP_TEXT = ”text”;

   private String text;

   public String getText() {
     return text;
   }

   public void setText(String text) {
     String oldText = this.text;
     this.text = text;
     propertyChangeSupport.firePropertyChange(
                                PROP_TEXT, oldText, text);
   }

...
```

Beans binding startades som JSR 295 men har sedan dess forkats till BetterBeansBinding. Vi kör vår egen patchade version av beans binding.

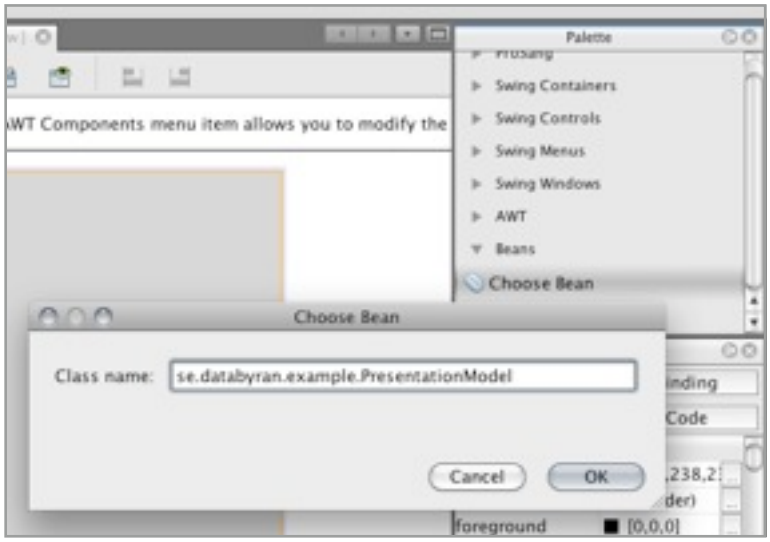
För att vara en property måste fältets getter heta get*FältetsNamn* (eller is*FältetsNamn* om det är en boolean) och settern set*Fältnamn*, du skall också deklarera en public strängkonstant som innehåller fältnamnet som heter PROP_FÄLTETS_NAMN så att externa klasser slipper hårdkoda fältets namn om de lyssnar efter ändringar av fältet.

Skärmritaren håller två filer i sync. En XML-fil som beskriver skärmen och en Java-klass med ett par segment som inte går att ändra genom NetBeans och som skärmritaren genererar från XML-filen.

Givetvis kan man koda med beansbinding utan att ha en skärmritare. Om du fäller ut den genererade koden så ser du hur det kan se ut.

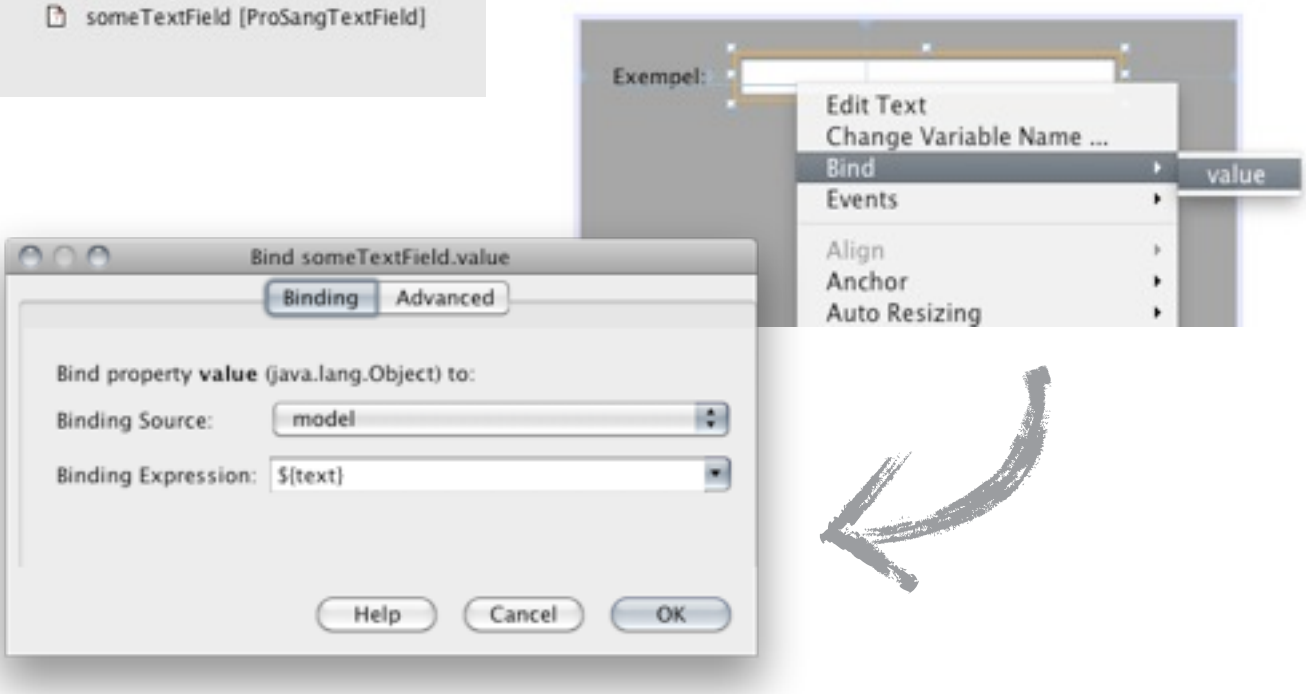
BeansBinding
Integration i NetBeans

Skärmritaren i NetBeans IDE har stöd för att skapa beansbinding bindningar mellan olika komponenter i skärmen (ett objekt kan vara en komponent i skärmen utan att synas).



För att skärmritaren skall hitta objekten måste de skapas genom skärmritaren. Det gör man genom att i skärmritaren välja ”Beans > Choose” Bean i paletten och sedan skriva hela klassnamnet på klassen man behöver ett objekt av. Därefter får man en markör för att placera komponenten i skärmen och då kan man klicka var man vill för att skapa komponenten.

När man gjort det kommer komponenten att skapas i den genererade koden och dyka upp i en vy som heter *inspector* i NetBeans som visar skärmens komponentträd.



Formstate revisited

Nu när vi har automagiskt synkronisering mellan presentationsmodellen och vår skärm kan vi dra ytterligare nytta av beansbinding genom ett speciellt formstate som använder beansbindings för att upptäcka när något i skärmen ändrats eller är felaktigt.

BeansBindingFormState tar en BindingGroup som argument till konstruktorn. Matisse skapar automatiskt en instans av BindingGroup så fort man skapat en bindning till någon egenskap för en komponent i skärmen.

BeansBindingFormState är också integrerat med hibernate validation så att bindningar i en skärm som har ett sådant formstate automatikst valideras on-the-fly och valideringsvarningar dyker upp som röda varningsikoner på skärmkomponenten som innehåller felet.

```
public AddressBookTopComponent() {
    initComponents();

    FormState formState = new BeansBindingFormState(bindingGroup);
    setFormState(formState);
}
```

ListState

Ett ListState håller reda på om något lagts till, tagits bort eller ändrats i en lista. En ProSangSimpleList innehåller automatiskt ett sådant liststate efter att den initialiserats. ListState är också en implementation av FormState vilket innebär att du i en skärm som bara innehåller en lista slipper ha ett eget formstate.

```
public AddressBookTopComponent() {
    initComponents();
    ...

    setFormState(mySimpleList.getListState());
}
```

Substates

Om du använder en FormState-implementation som heter DefaultFormState kan du lägga till ett träd av andra formstates som substates till det. Om någon av substatesen är felaktig eller ändrad så bubblar detta upp till vårt formstate. Bra för komplexa skärmar.

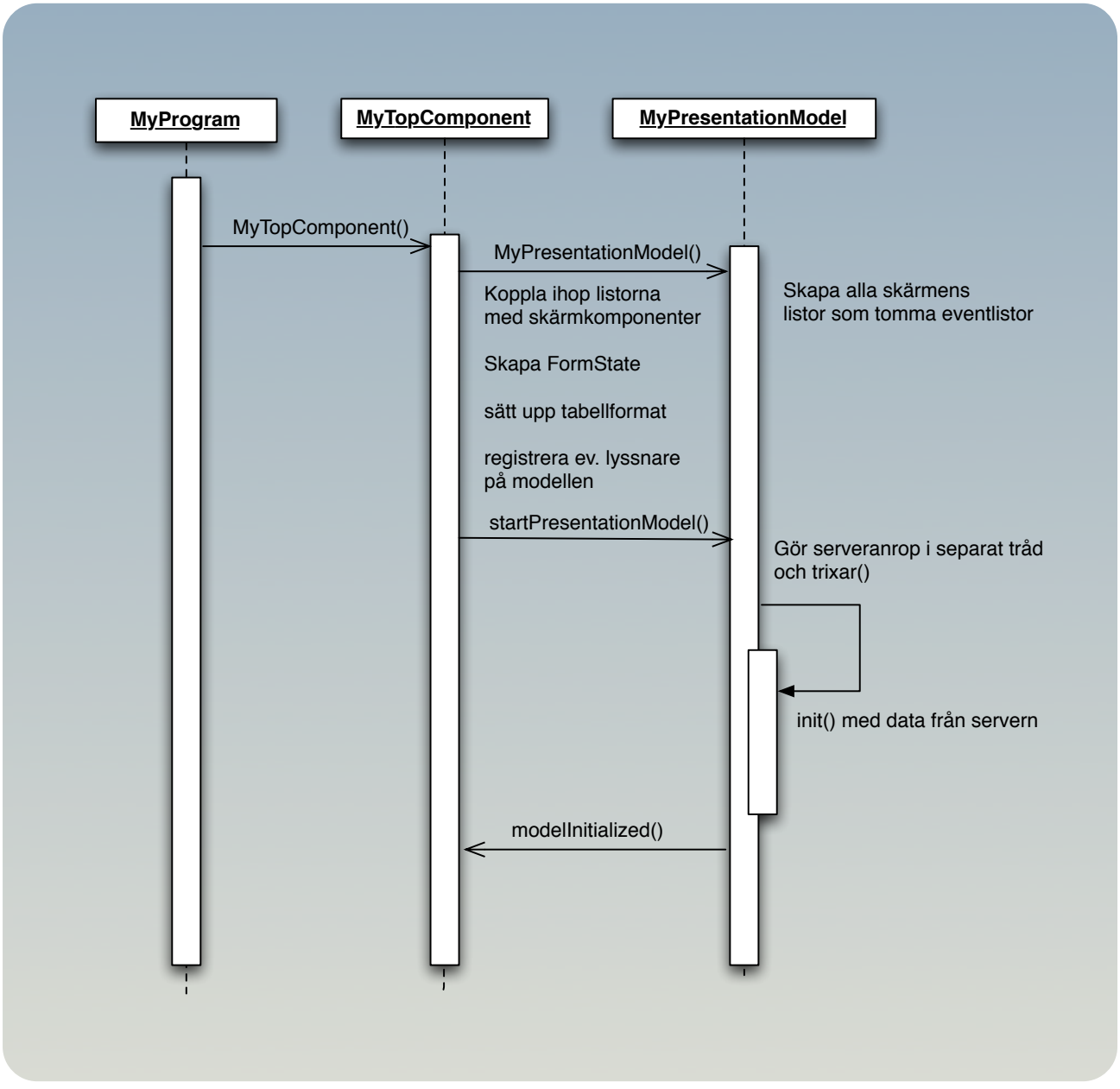
```
public AddressBookTopComponent() {
    initComponents();
    ...
    DefaultFormState formState = new DefaultFormState(”Main”); // NOI18N
    formState.addSubState(new BeansBindingFormState(bindingGrou));
    formState.addSubState(mySimpleList.getListState());
    setFormState(formState);
}
```

Skärmens livscykel

Eftersom det går att skapa skärmar på tre tusen sätt är det viktigt att topkomponenterna har en livscykel som är likadan för alla skärmar så att det är lätt att förstå hur en skärm fungerar.

Därför skall du implementera laddning så här:

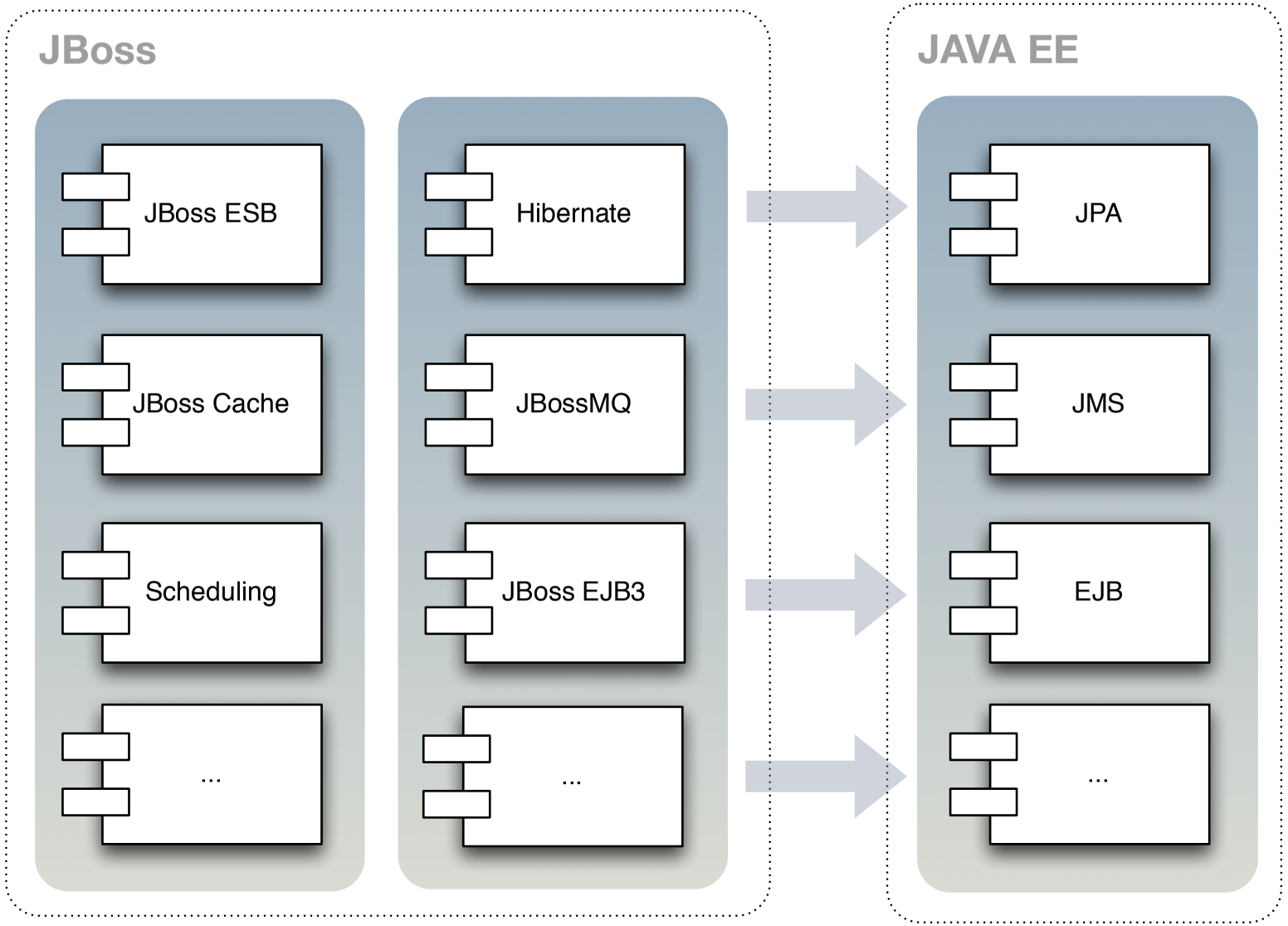
När skärmen skapas



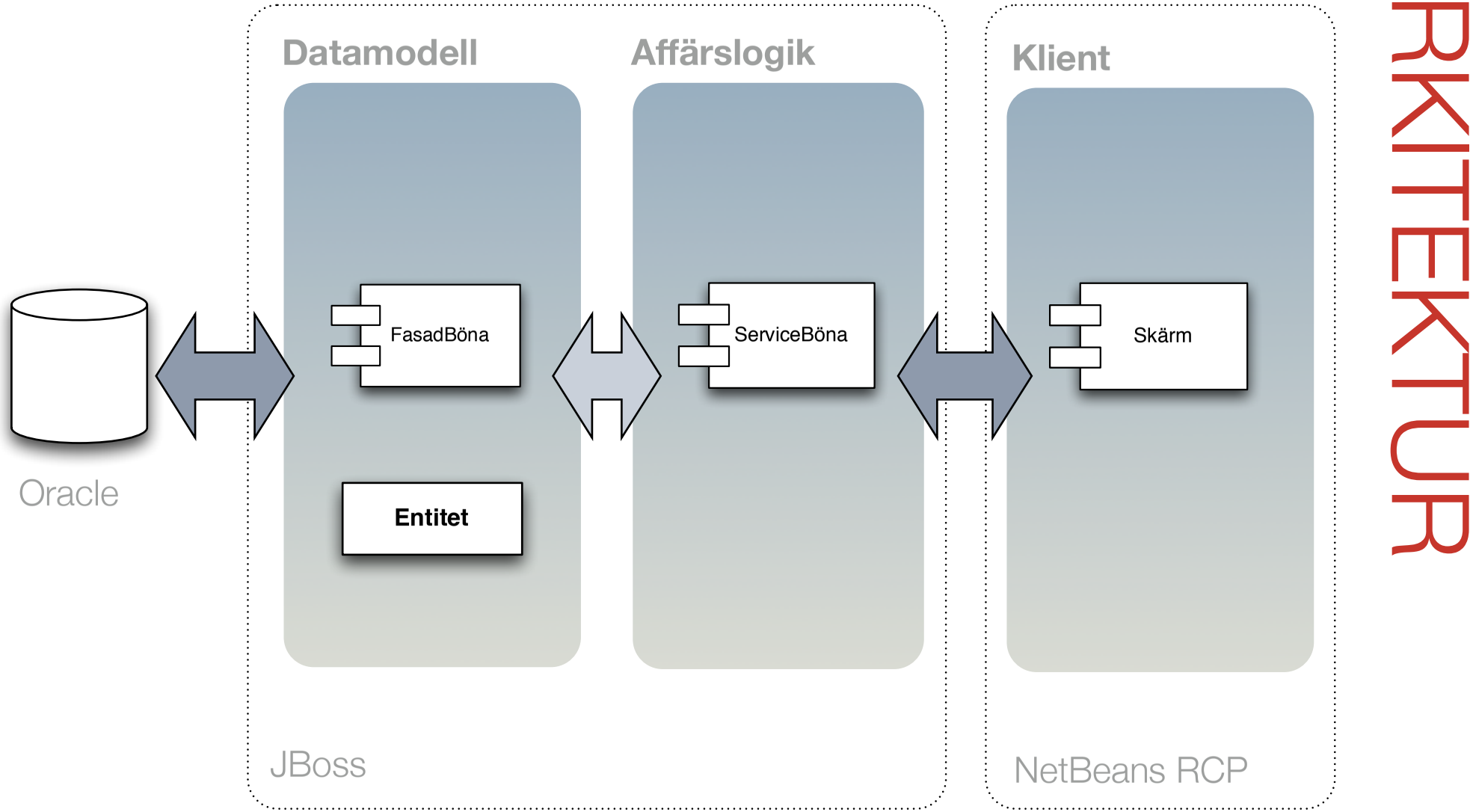
När användaren sedan är klar med skärmen och klickar Ok för att spara sina ändringar skall skärmen alltid spara med hjälp av FinalServerOperation och PresentationModel.performFinal().

Presentationsmodellen kommer då att skapa en transaktion från klienten så att flera anrop till servern knyts ihop till en transaktion samt ha möjlighet att spara transaktionslog och annat som är generellt för alla skärmar i hela ProSang.

JAVA EE


JBoss
JBoss ESB
JBoss Cache
Scheduling
...
Hibernate
JBossMQ
JBoss EJB3
...
JAVA EE
JPA
JMS
EJB
...

ARKITEKTUR


Datamodell
Affärslogik
Klient
FasadBöna
ServiceBöna
Skärm
Entitet
Oracle
JBoss
NetBeans RCP

DATAMODELL

Entiteter

databas + klasser = sant

Ett klassiskt problem i objekt-orienterade program som jobbar mot en databas är hur skall relationsdatat flyttas fram och tillbaka mellan objekt och databasens tabeller.

Vi använder JPA för detta (Hibernate närmare bestämt).

JPA går ut på att man mappar klasser mot tabeller och relationsdatabasstrukturen. För att kunna använda JPA krävs att man för varje post i varje tabell som skall mappas har ett unikt id. I alla nya tabeller använder vi ett helt syntetiskt id från en oraclesekvens.

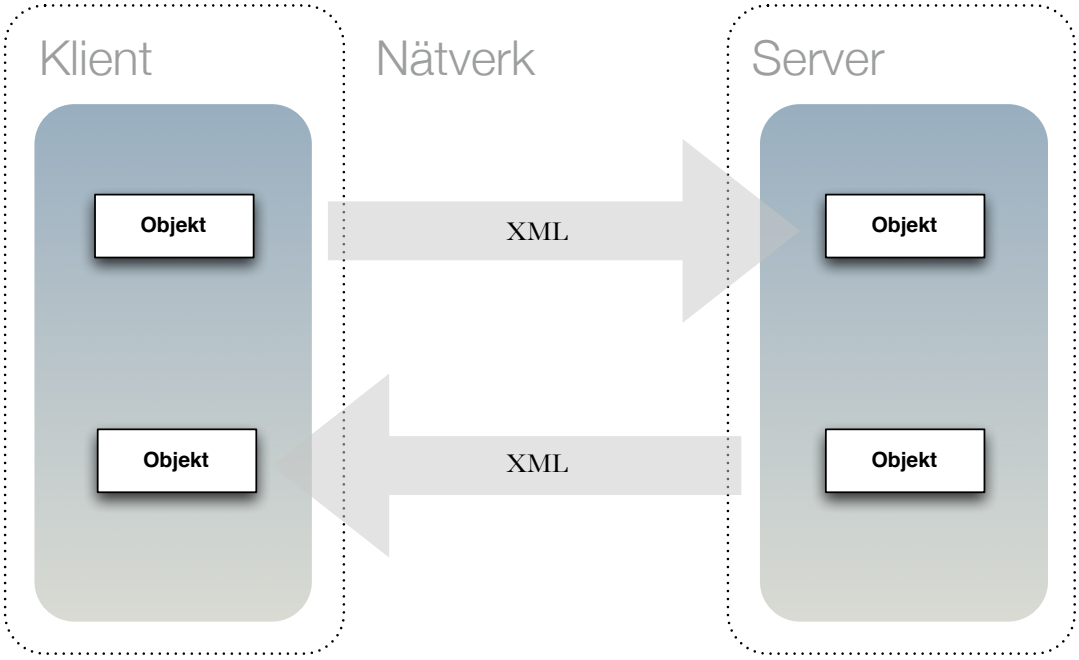
```
@Entity
@Table(name = ”ADDRESS”)
public class Address {

       @Id
       @Column(name = ”ID”)
       @GeneratedValue(strategy = GenerationType.AUTO)
       private long id;


      @Column(name = ”NAME”)
      private String name;


...
```

För att jobba mot databasen använder man sedan en EntityManager som innehåller metoder för att läsa, skriva, ta bort och söka efter entiteter. Man använder ett speciellt query-språk som heter JPQL för att söka efter entiteter som innehåller en del finesser som inte finns i SQL men i övrigt är ganska snarlikt.

```
EntityManager em = ...

Query query = em.createQuery(”SELECT a FROM Address a”);
List<Address> allAddresses = query.getResultList();

query = em.createQuery(”SELECT a FROM Address a ” +
                       ”WHERE a.name = :name”);
query.setParameter(”name”, ”Testa Testsson”);
Address result = query.getSingleResult();
```

Serializable

En inbyggd finess i Java är förmågan att serialisera en objektgraf till XML eller tvärtom. Finessen används av Java RMI (Remote Method Invocation) som i sin tur är det sätt en klient pratar med en Java EE server över nätverket.



För att ett objekt av en klass skall gå att serialisera måste det implementera ett markup interface som heter Serializable. Det måste också ha ett versionsfält för att veta att klassen som objektet serialiserades ifrån är likadan som klassen som objektet avserialiseras till. Versionsnummret måste uppdateras varje gång du ändrar entitetens fält på något sätt. Om du ändrar metoderna behöver du inte räkna upp versionsnummret.

Att klassen är serializable är inte direkt kopplat till just nätverkskommunikation utan innebär att objekten kan skrivas som XML till vilken ström som helst. T.ex. för att skriva en lista med objekt till en fil på hårddisken.

```
public class Address implements Serializable {

    private static final long serialVersionUID = 1L;

...
```

PersistenceContext
Attached/Detached

En mycket viktig skillnad mellan JPA och att läsa och skriva data med SQL är begreppet attached. JPA håller ett cache som lever under hela transaktionen. Om samma entitet läses flera gånger kommer alla referenser som man får ut peka på samma instans av entiteten.

Så länge entiteten är attached kommer också alla ändringar som görs på den att skrivas tillbaka till databasen när transaktionen committas.

```
...
EntityManager em = ...
em.getTransaction().begin();

Address address1 = em.find(Address.class, 1);
Address address2 = em.find(Address.class, 1);
// address1 == address2

address1.setName(”Nytt namn”);
em.getTransaction().commit();
```

Det innebär alltså att man inte kan läsa ut en entitet och sedan använda den som någon form av temporärlagring (varför man nu skulle göra det) eftersom de ändringar man gör alltid skrivs tillbaka till databasen.

Om man sparar en ny entitet med EntityManager.persist så kommer objektet man skickade in till persist att vara attached efter anropet. Om man sparar en gammal entitet med EntityManager.merge så kommer objektet man skickade in till merge inte att vara attached, däremot så kommer det sparade och attachade objekt returneras.

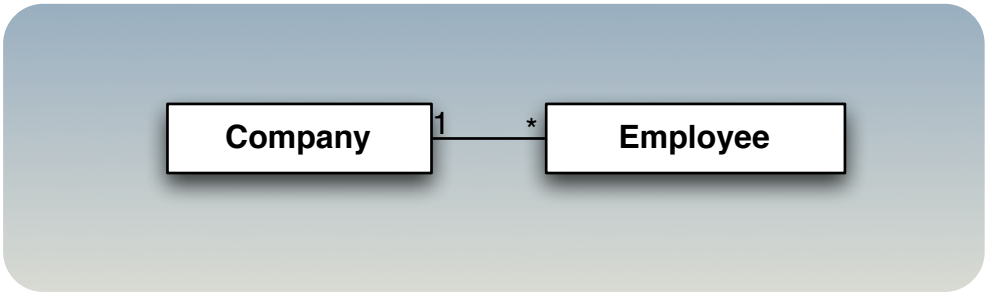
```
EntityManager em = ...
em.getTransaction().begin();

Address address1 = new Address();
em.persist(address1);

address1.setName(”Nytt namn”);
em.getTransaction().commit();
```

När ett objekt serialiseras så kommer det automatiskt att bli detached. Objekten serialiseras alltid när de skickas fram och tillbaka mellan server och klient.

Relationer
LazyInitializationException

Halva poängen med relationsdatabas är ju att ha relationer mellan olika tabeller. Att varje gång man laddar en entitet läsa ut alla entiteter den har relationer med är inte så bra därför lazy-laddas de flesta relationer i ProSang.

Om man inte annoterar en relation med något så tolkar hibernate det som att relationen skall lazy-laddas.




```
@Entity
public class Company {

    @OneToMany
    private List<Employee> employees = new ArrayList();

...
```

Så länge en entitet är attached så laddar hibernate automagiskt relationer först när man faktiskt använder dem. Om enteteten är detached kan har inte hibernate längre någon kontakt med enteteten och man får därför en LazyInitializationException. Det innebär att alla relationer som behövs på en entitet måste laddas innan den skickas till klienten.

```
EntityManager em = ...
...
Query query = em.createQuery("SELECT c FROM Company c " +
                             "WHERE c.name = :name");
query.setParameter("name", "Databyrån");
Company company = query.getSingleResult();

List<Employee> employees = company.getEmployees();

// lazy load does not happen until here
int numberOfEmployees = employees.size();
```

Transaktioner
JavaEE/JavaSE

När man använder JPA i JavaSE måste man sköta start, commit och rollback av transaktionen själv. I våra tester använder vi JPA i JavaSE miljö men för att slippa tänka på transaktionerna så använder vi ett lite färdiga finesser från Jee5Unit för att få en ny transaktion i varje test och en rollback efter varje test.

Om man kodade en JPA-applikation på JavaSE skulle man få lov att göra:

```
EntityManager em = ...
em.getTransaction().begin();

...

em.getTransaction().commit();
em.getTransaction().rollback();
```

I JavaEE sköts transaktionerna av applikationsservern om vi inte annoterar våra EJB:er på ett speciellt sätt. Applikationsservern rullar tillbaka transaktionen om en EJB-metod kastar ett exception. För många skärmar använder vi nu en speciell ServerOperation som startar en transaktion från klienten och knyter ihop flera serveranrop i samma transaktion.

Man kan styra hur EJB-metoderna vill ha sina transaktioner med annoteringar men vi gör sällan det.

Optimistisk låsning
Skydd mot samtidiga ändringar

Om två användare läser ut samma databaspost och sedan sparar ändringar så kommer den som sparar sist skriva över den första användarens ändringar utan att någon märker något.

För att komma runt problemet finns två lösningar.

Pessimistisk låsning innebär att den första användaren som läser posten låser den så att ingen annan kan läsa posten förrän användaren är klar och låst upp posten.

Optimistisk låsning innebär att man på något sätt håller koll på hur många gånger en post skrivits eller när en databas post senast skrevs. Innan man skriver tillbaka sina ändringar jämför man sin post med den i databasen, har den samma antal ändringar alternativt samma ändringsdatum? Då har ingen ändrat posten och det går bra att uppdatera. Om antalet skrivningar räknats upp eller datumet ändrats till ett senare datum så har någon annan gjort ändringar och vi måste avbryta skrivningen.

I JPA finns stöd för både datum och versions-lösningen. Vi använder versionslösninen. Alla nya tabeller i ProSang har alltså en versionskolumn som räknas upp varje gång en post ändrats.

```
@Entity
@Table(name = ”ADDRESS”)
public class Address implements Serializable {
   ...

   @Version
   private long optlock;


...
```

På något ställe i ProSang använder vi pessimistisk låsning. T.ex. för att garantera att det inte blir hål i tappningsnummerserierna.

I praktiken så skapar vi nästan aldrig versionsfältet själva utan får det från en basklass som heter DefaultEntity (som också ser till att alla entiteter har ett long-id, skapad-data och senast-ändrad-data och en propertyChangeSupport).

```
@Entity
@Table(name = ”ADDRESS”)
public class Address extends DefaultEntity
                     implements Serializable {


...
```

Unittester av datamodellen
Veta att det fungerar utan deploy

För att kunna testa att datamodellens klasser är rätt mappade, att queryn fungerar så använder vi en SQL-databas som skapas av ett testframework som heter Jee5Unit. SQL-databasen skapas i arbetsminnet på din dator då det första entitetstestet körs och kastas sedan bort när alla tester är klara.

Ovanpå Jee5Unit har vi skapat ett par basklasser för test som sätter upp lite extra data för ProSang (bland annat en fejkad inloggad användare).

Unittester som testar en entitet genom att läsa och skriva den till databas skall heta Klassnamn + EntityTest.

```
public class AddressEntityTest extends EntityTest {

   @Test
   public void testIsPersitableAndReadable() {
       EntityManager em = getEntityManager();

       Address instance = new Address();
       instance.setName(”Nisse”);
       em.persist(instance);

       em.flush();
       em.clear();

       Address persisted = em.find(Address.class,
                                   instance.getId());

       assertNotNull(persisted);
  }

...
```

Jee5Unit är skrivet av Johan. Mer info och exempel på hur man kan använda det finns på projektsiten: http://markatta.com/jee5unit/

Eftersom vi inte kan göra commit på transaktionen (och potentiellt göra så att andra tester slutar fungera) så får vi lov att göra lite knep. EntityManager.flush() tvingar Hibernate att göra om våra ändringar till SQL och trycka dem ut till databastransaktionen, på så sätt fångar vi upp eventuella exceptions som är en följd av att vi mappat entiteten felaktigt på något sätt.

EntityManager.clear() tar bort alla objekt i persistence-contexten så att vi är säkra på att när vi sedan laddar den så laddas den från databasen. Om vi inte gjorde det skulle EntityManager.find() i exemplet ovan bara returnera en referens till instance.

Validering

För att kunna lita på databasens innehåll behöver allt data man får in valideras. Istället för att koda sådana krav på alla platser data kan komma in i ProSang använder vi ett valideringsframework som heter Hibernate Validation och som är tätt knutet till Hibernate.

På entiteternas fält kan man annotera krav som kommer att undersökas automatiskt i skärmarna där beans binding används. Fältens data matchas också mot kraven innan en entitet skrivs till databasen. Om något fält är felaktigt när databasskrivningen skall ske kastar Hibernate ett exception som rullar tillbaka transaktionen. Det blir alltså i praktiken omöjligt att spara felaktigt data genom ProSang.

Fältkraven annoteras till skillnad från entitetsannoteringarna på fältets get-metod.

```
@Entity
public class Address extends DefaultEntity {

    @Required
    private String name;

    @Length(max = 255)
    @Column(name = ”NAME”)
    public String getName() {
        return name;
    }

    ...
```

Annotering	Beskrivning
@Required	Fältet måste ha ett värde - får inte vara tomt
@Length(min = , max =)	Krav på hur många tecken som får vara i en sträng
@Max(value=)	Krav på hur högt ett tal i en sträng eller ett numeriskt fält får vara
@Min(value=)	Samma som ovan fast minsta värde

Läs om fler inbyggda valideringsannoteringar här:
http://docs.jboss.org/hibernate/validator/3.x/reference/en/html/validator-defineconstraints.html

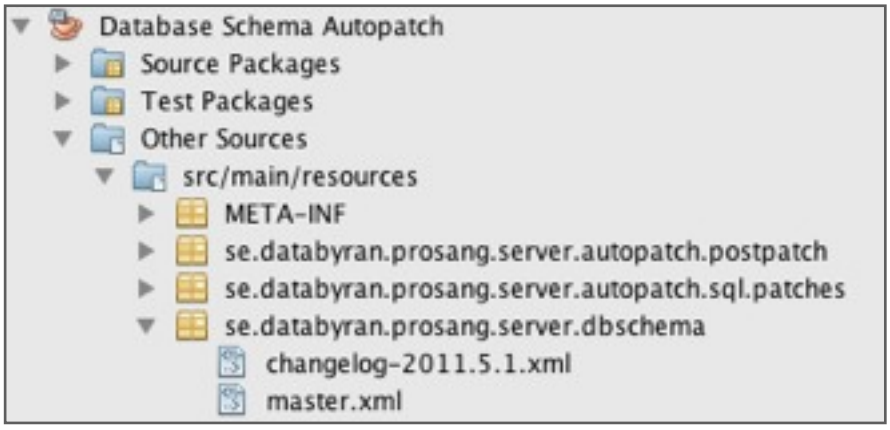
Databasschemat
XML-representation

ProSang har vid varje version ett exakt databasschema som matchar de entiteter som finns i datamodellen. För att garantera att det är så använder vi ett opensource-bibliotek som heter LiquiBase

Lösningen fungerar såhär: varje gång en databastabell skall skapas eller förändras skapar man ett nytt ”changeset” i XML som beskriver en eller flera förändringar av databasschemat på ett databasoberoende sätt. Changeset:et har ett unikt id som består av ditt användarnamn samt ett strängid och xml-filen som det ligger i. Med hjälp av det håller liquibase koll på exakt vilka ändringar som applicerats och vilka som är nya.

Liquibase kan också rulla tillbaka (många) ändringar i efterhand så att det är möjligt att köra sin ändring, upptäcka att man gjort fel, rulla tillbaka och rätta sin ändring när man sitter och utvecklar.

För att hjälpa till då man skapar changesets har vi ett eget netbeansplugin som gör att man kan högerklicka på en entitet och få ut ett LiquiBase-changeset som beskriver hur tabellen skall se ut.



Mer information om pluginet hittar du på wikin: http://wiki.databyran.se/display/psdev/Databasschemahantering

För att säkerställa att schemauppdateringen gått klart innan ProSang startar så körs schemauppdateringen med hjälp av egna JBoss-services som sedan alla fasadbönor har ett beroende på.

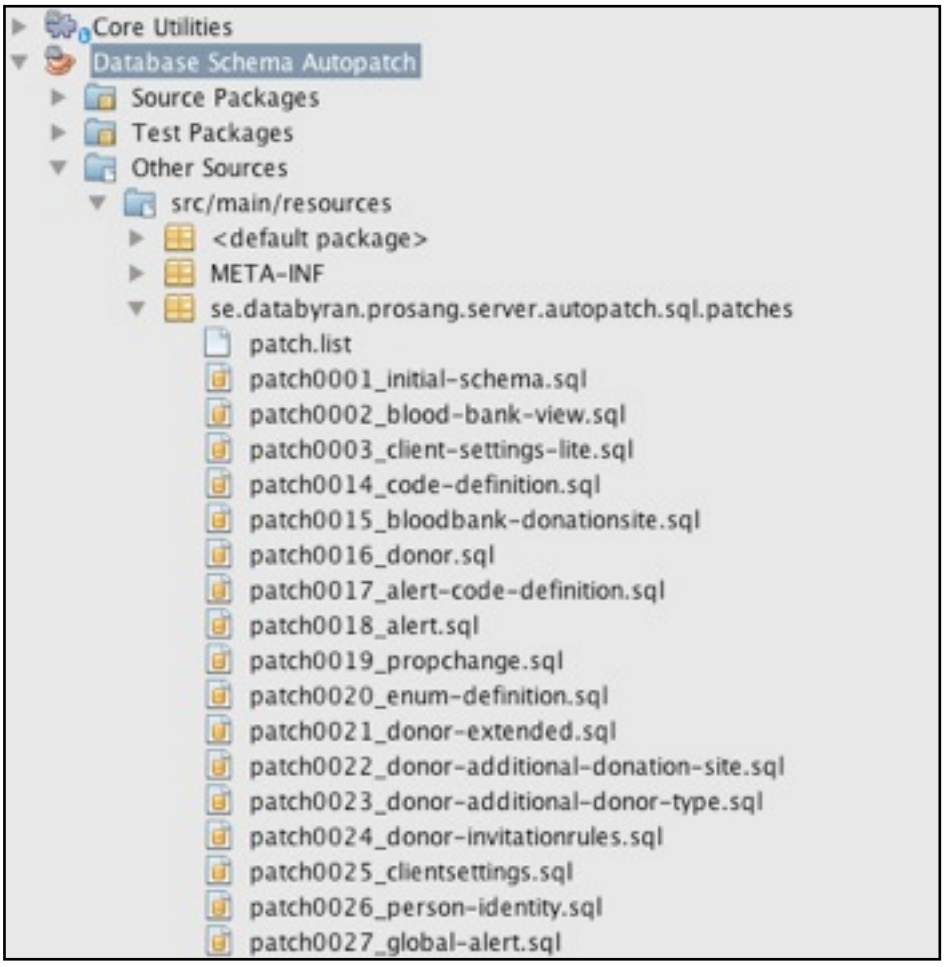
Databasschemat

Deprecated:Generera DDL från klasser

Det här gäller ProSang före version 2011.5 då vi gick över till LiquiBase (se föregående sida). Det är kvarlämnat som en referens.

ProSang har vid varje version ett exakt databasschema som matchar de entiteter som finns i datamodellen. Vi kan generera DDL från annoteringarna i våra klasser men vi kan inte applicera dem automatiskt. En del databasschemadetaljer som t.ex. index finns dessutom inte alls i entiteterna och måste läggas till för hand.

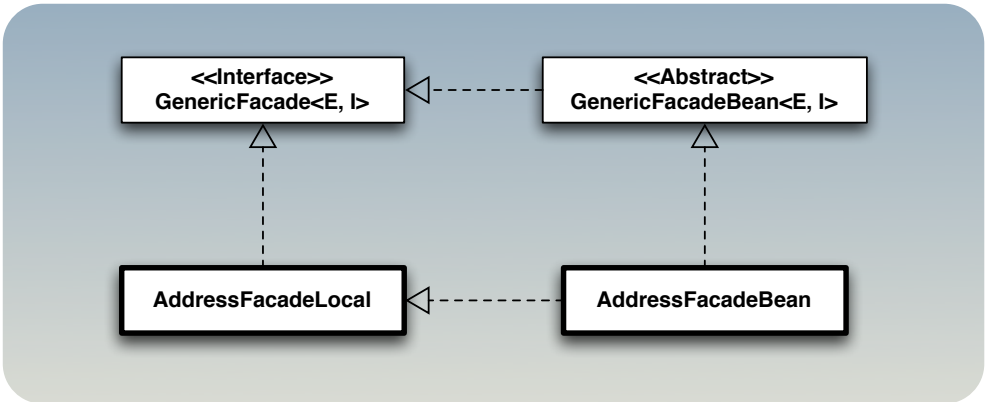
Lösningen fungerar såhär: varje gång en databastabell skall skapas eller förändras skapar man en ny databasschemapatch som placeras i en speciell katalog. Nästa gång ProSang deployas på servern kommer patchen köras. Ett speciellt bibliotek håller reda på vilka patchar som gått och vilka som är nya så



För att generera databasschema från datamodellen högerklickar du på datamodellsprojektet, väljer ”Custom > Generate DDL”, maven kommer då generera en DDL för hela datamodellen som dels skrivs ut i NetBeans logfönster och dels hamnar i en fil.

Du får själv leta fram de ändringar du skall ha i din patch.

Fasadbönor
Läsa och skriva entiteter utifrån



För att undvika att logik som läser och skriver entiteter sprids genom hela ProSangservern har varje viktig entitet en fasadböna som resten av ProSang använder för att läsa och skriva entiteter av den typen. Eftersom JPA:s queryspråk inte är typsäkert ger det också ett extra skydd vid refaktorering - det är hyffsat lätt att hitta alla JPQL-queries som görs mot en specifik entitet.

Ett antal operationer (CRUD) är generella och finns färdigimplementerade i en abstrakt basklass som heter GenericFacadeLocal. Metodsignaturerna för dessa finns också i ett generellt interface som du låter ditt EJB-interface uttöka.

Krångligare operationer som sökningar t.ex. ges en egen metod i fasad-EJB:n.

Fasad-EJB:erna är alltid bara lokala och kan inte anropas från en extern klient till ProSangservern.

```
@Local
public interface AddressFacadeLocal extends GenericFacade<Address, Long> {
   List<Address> findSpecialAddresses();
}

@Stateless
public class AddressFacadeBean extends GenericFacadeBean<Address, Long>
                               implements AddressFacadeLocal {
   public List<Address> findSpecialAddresses() {
      ...
   }
}
```

AFFÄRSLOGIK

Affärslogiken är det som knyter ihop flera fasadböneanrop till affärsmetoder som publiceras till klienten. Ibland kan affärslogiksbönorna till en början vara enkla delegat som bara upprepar metoder från fasadbönorna. Ofta kan det vara bra att knyta ihop upprepade anrop av en fasadböna till ett enda anrop på en affärslogiksböna.

Fasadbönorna som en affärslogiksböna använder injiceras automatiskt av applikationsservern när vi annoterar dem med @EJB

Alla våra affärslogiks EJB:er är remote-bönor och måste därför annoteras med en behörighetsannotering som styr vem som får anropa dem. Med några få undantag kräver alla våra remote-EJB:er att man klienten är inloggad och annoteras därför med krav på rollen ”user” (om man inte annoterar sin EJB med behörighet kan vem som helst ropa på metoderna i den).

```
@Remote
public interface AddressRemote {
   /**
    * Save all addresses in the list and return true
    * if the save was successful
    */
   boolean saveAddresses(List<Address> addresses);
}
```

```
@Stateless
@RolesAllowed("user")
public class AddressBean implements AddressRemote {

    @EJB
    private AddressFacadeLocal addressFacade;

    public boolean saveAddresses(List<Address> addresses) {
        for (Address address : addresses) {
            if (address.getId() == 0) {
                addressFacade.insert(address);
            } else {
                addressFacade.update(address);
            }
         }
         return true;
    }
}
```

Unittester av affärslogiken
Testa utan datamodellen

När man testar affärslogiken så vill man inte att testerna skall fela på grund av ett fel i en fasadböna. Man vill faktiskt bara testa just den logik man skrivit i sin affärsmetod.

För att åstadkomma ett sådant test använder vi ett mock-framework som heter JMock som fungerar lite som en skådespelare. Man ger ett interface till JMock och säger, låtsas att du är en implementation av det här interfaces. Sedan ger man ett manus (Expectations) till JMock där man beskriver hur man förväntar sig att någon kommer använda implementationen.

För att få sin skådespelarklass injicerad i sin EJB-klass som om den var deployad på en applikationsserver låter vi vårt test ärva EJBTestCase och använder metoderna injectEJB() samt getBeanToTest().

```
public class AddressBeanTest extends EJBTestCase<AddressBean> {

    public void AddressBeanTest {
        super(AddressBean.class);
    }

    @Test
    public void testSaveAll() {
        Mockery mockery = new Mockery();

        final AddressFacadeLocal mockedAddressFacade =
                       mockery.mock(AddressFacadeLocal.class);

        List<Address> addresses = new ArrayList();
        addresses.add(new Address());
        addresses.add(new Address());

        mockery.checking(new Expectations() {
            {
                exactly(2).of(mockedAddressFacade).
                                        save(with(any(Address.class));
                will(returnValue(Boolean.TRUE));
            }
        }

         injectEJB(AddressFacadeLocal.class, mockedAddressFacade);
        AddressBean instance = getBeanToTest();

        boolean result = instance.save(addresses);
        assertTrue("Expected save to return true on success", result);
        mockery.assertIsSatisfied();
   }

...
```

UTSKRIFTER

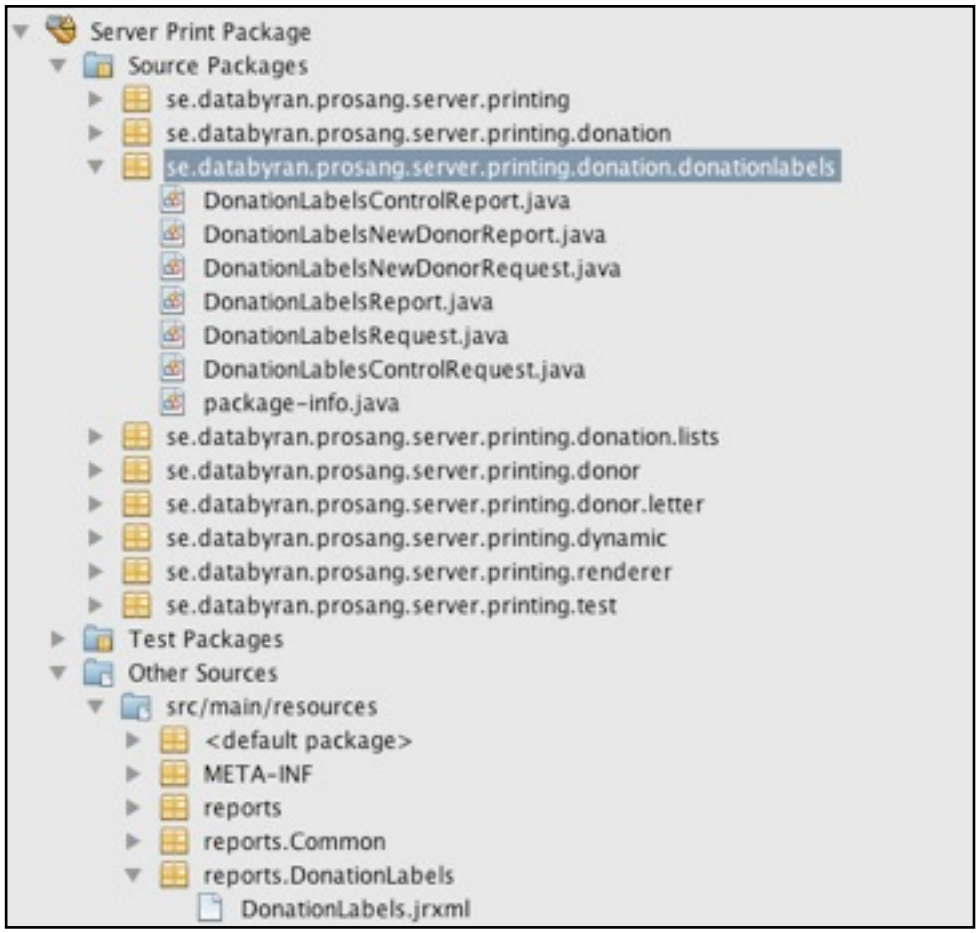
ProSangs utskriftspaket använder iReports/JasperReports som är ett rapportverktyg som är integrerat med NetBeans. Man ritar alltså sina utskrifter inuti NetBeans.

Kring utskrifter finns ett par viktiga begrepp:

- **rapporttyp** en klass som innehåller logiken som tar fram och förbereder data för rapporten, klassen innehåller också en lista med de rapporter som man kan använda för att skriva ut rapporttypen
- **rapport** själva layouten för en utskrift, kan finnas flera för en rapporttyp
- **utskrift** data från en rapportklass kombinerat med en rapport färdigt för att skriva ut.
- **ReportRequest** knyter ihop en rapporttyp, vilken rapport som skall användas samt eventuella parametrar till rapporttypen (t.ex. vilken givare som skall skrivas ut)

Klienten ropar på PrintRemote med en ReportRequest och servern generar sedan utskriften som den skickar tillbaka till klienten så att utskriften köas genom klientendatorns utskriftssystem. Servern känner alltså inte till något om vilken skrivare som används.

Rapporttyperna och rapporterna finns i modulen Server Print Package.



TRANSAKTIONSLOG

I ProSang behövs väldigt finkornig spårbarhet på ändringar. Alla operationer som måste vara spårbara skrivs till transaktionsloggen. Operationer som skall var spårbara kan vara att ett visst fält på en entitet ändrats, att en rapport skrivits ut eller att användaren forcerat någon form av varning.

Datamodellen

Entiteter kan själva hålla reda på om vissa fält ändrats. Genom att sedan låta dem implementera SelfLoggingEntity kan man skicka själva instansen till fasadbönan LogFacade. Man kan t.ex. göra det från insert och update på fasadbönan för entiteten.

Den här typen av loggning är enklast att implementera för rot-entiteter som man alltid jobbar direkt med. Entiteter som sparas med cascade är svårt att hantera på detta sätt.

Affärslogiken och fasadbönorna

Man kan skapa LogItems direkt i sin kod på servern och skicka till LogFacade.

Klienten

I klienten vill man oftast bara spara LogItems om användaren sparar sina ändringar. Därför är det bäst att skapa en LogHandler i sin presentationsmodell som man sedan ger LogItems när saker som skall transaktionsloggas händer. När sedan skärmen kör sin FinalServerOperation så ger man listan med logitems från sin LogHandler till sin FinalServerOperation så sparas de automatiskt.

GUITESTER

ProSang har funktionstester som testar hela applikationen från användargränsnitt i klienten ner till databasen genom en deployad server. Testfallen skrivs som vanliga unittester i modulen `Client Modules/Client Application`.

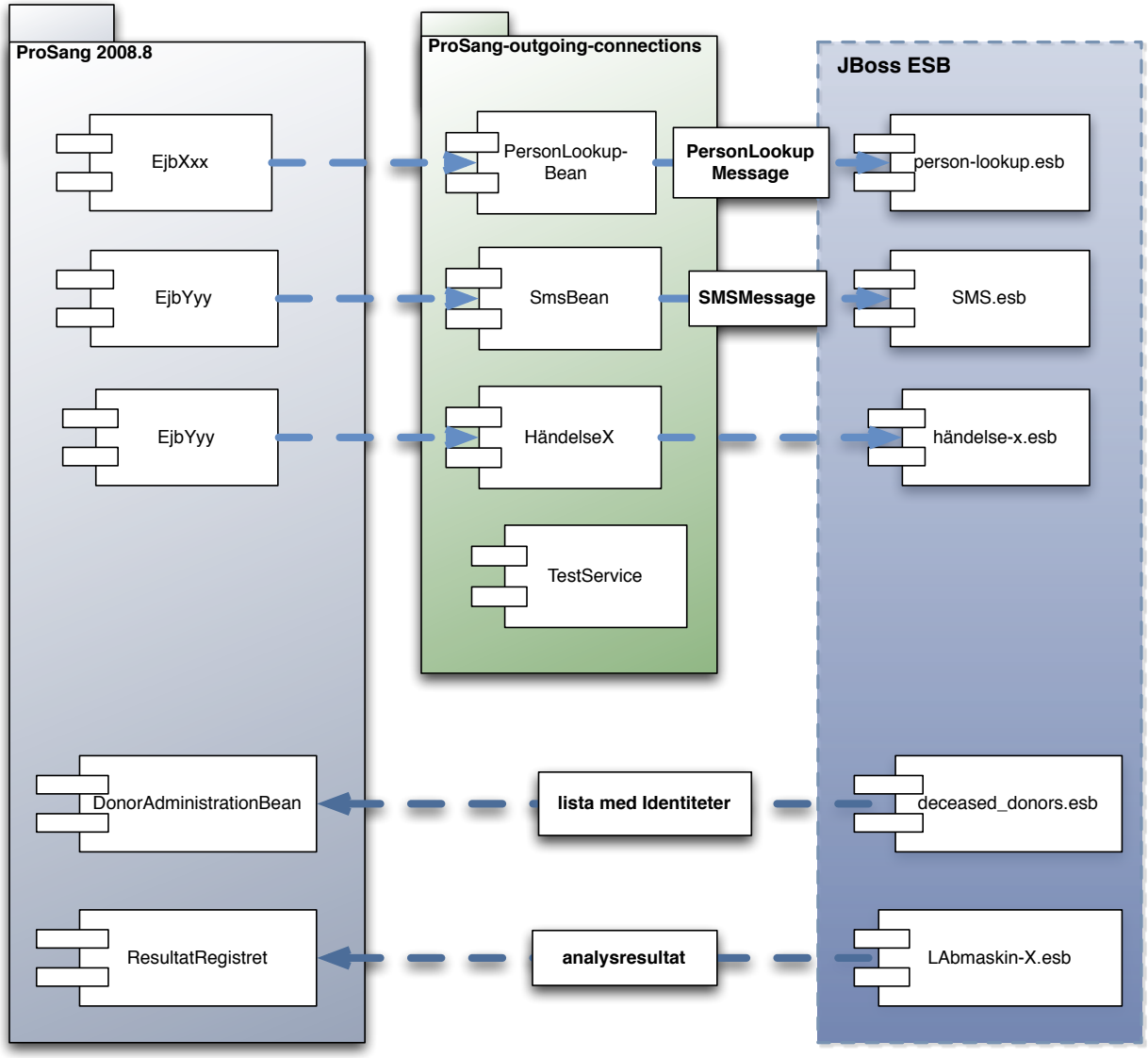
Testerna fungerar som om det var en användare som körde systemet.

För varje swing-komponent finns en Operator-klass som vet hur man jobbar med en sådan komponent.

JBOSSESB

Hos varje kund är förutsättningarna för integration med andra system och indatakällor olika. Det kan t.e.x. vara vad kunden köper för typ av utgående SMS-tjänst, olika SMS leverantörer har helt olika sätt att skicka SMS. Någon har en webtjänst, någon har en HTTP-server som tar meddelandet som GET argument i en URL.

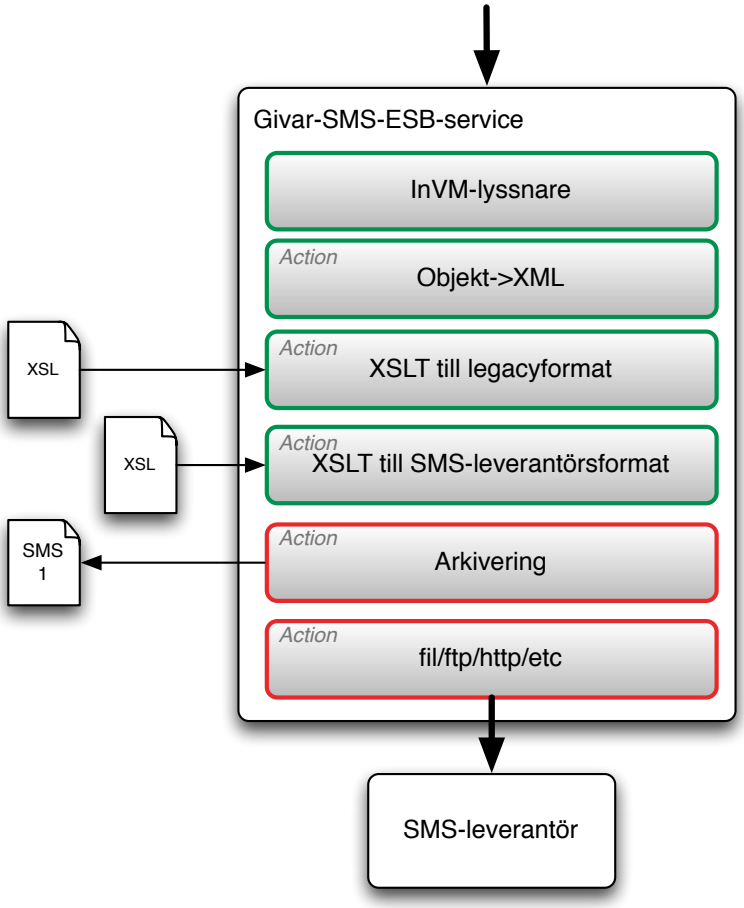
För att slippa göra en version av ProSang per kund är denna integration externaliserad med hjälp av JBossESB (JBoss Enterprise Service Bus). Varje ingående och utgående datakoppling är en ”tjänst” som deployas på JBoss-servern. ESB-tjänsterna kataloger istället för jar-arkiv och läggs precis som allt annat som deployas på en JBoss-server i deploy-katalogen.

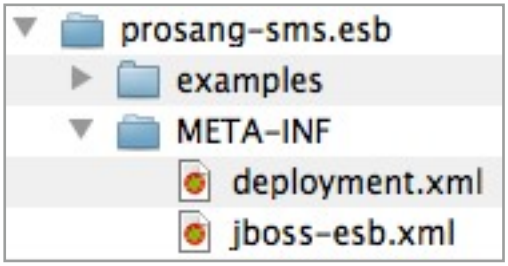


ESB-TJÄNST

En ESB-tjänst börjar med en gateway där data kommer in på något sätt. Det kan vara en webbtjänst, en katalog där filer placeras, en plats på en FTP-server där filer placeras, m.m. Man kan också explicit anropa en viss ESB från sin javakod. Alla sådana anrop sker i ProSang ifrån lokala EJB:er i prosang-outgoing-connections.jar så att själva ProSang inte behöver känna till JBossESB på något sätt.

Efter detta består ESB-tjänsten av en eller flera Actions som skall bearbeta datat eller publicera datat någonstans. Exempel på en action kan vara XSLT-transformation av XML, att skriva till ett loggningsarkiv eller använda datat som argument till ett EJB-anrop. Ett antal färdiga ESB-actions är inbyggda i JBoss ESB och vi har ett eget bibliotek med några egna actions. I första hand använder ESB-tjänster färdiga actions som konfigureras på olika sätt.





En ESB-tjänst består i filsytemet av en katalog som slutar på .esb i den finns en META-INF-katalog som i sin tur innehåller filen jboss-esb.xml som innehåller tjälva tjänstekonfigurationen och ibland deployment.xml som listar tjänstens beroenden på andra typer av tjänster i samma server.

JBOSS-ESB.XML

Det här exemplet är ProSangs utgående SMS-koppling. Attributet invmScope säger åt JBoss ESB att skapa en ingående kanal för anrop från samma JVM, det är den kanalen vi ropar på från prosang-outgoing-connections.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<jbossesb
    xmlns="http://anonsvn.labs.jboss.com/labs/jbossesb/trunk/product/etc/schemas/xml/
jbossesb-1.0.1.xsd"
    parameterReloadSecs="5">
    <!--
       This is an example format that just writes the SMS request to an archive in
       the tmpdir of the machine the server runs on.
       This is meant for testing on devel machines. See examples/ for actual
        production configuration samples.
       -->
    <services>
        <service category="ProSang"
                 name="SMS"
                 description="ProSang outgoing SMS service"
                 invmScope="GLOBAL">

            <!-- the mep attribute is what JBoss ESB looks at to know if we
                 extect some kind of status/answer when the service has handled
                 a message -->
            <actions mep="OneWay">

                <action name="debug" class="org.jboss.soa.esb.actions.SystemPrintln">
                    <property name="printfull" value="false" />
                    <property name="message" value="Value from getMessage" />
                </action>


                <!-- Transform the ProSang-Java SMS object into XML -->
                <action name="transformToXML"
                        class="org.jboss.soa.esb.actions.converters.ObjectToXStream">
                    <property name="exclude-package" value="true" />
                </action>




                <!-- archive message under your tmp directory -->
                <action name="archive"
                        class="se.databyran.esb.actions.ArchiveWiretap">
                    <property name="baseDirectory" value="${ProSangHome}/archive"/>
                    <property name="serviceName" value="outgoing-sms"/>
                </action>
            </actions>
        </service>
    </services>
</jbossesb>
```

API-ÖVERSIKT

I ProSang finns sjutusen olika små API:er för olika saker. Här är några:

Finder-API

API för sökdialoger som används för att välja t.ex. givare att öppna i ett program eller tillsammans med ProSangFinderField i en skärm där man vill kunna välja en entitet bland många. Ligger i Client Modules/Core Cluster/Core Search API. Exempel på implementationer finns i Client Modules/Donor Cluster/Donor Common

Sökparameter-API

API för att representera ett JQPL-sökargument med en klass som kan kombineras med andra sökargument som söker efter samma typ av entitet. Ligger i datamodellen i paketet se.databyran.prosang.model.core.search och exempel på implementationer finns i se.databyran.prosang.model.donor.search.

Matchat med detta finns också ett sökpanels-API i klienten som gör att man enkelt kan skapa ett program som tillåter användaren att kombinera enskilda sökparametrar till mer komplexa sökningar. Det kan du hitta i projektet Client Modules/Core Cluster/Core Dynamic Search API

Notifierings-API

API för att visa att en Swing-komponent innehåller felaktiga värden. Ligger i ProSang Libraries/ProSang GUI-lib där både API och ett antal implementationer ligger. Innehåller interfacet Notifiable som BeansBindingFormState använder för att varna om fel och som du också använder om du vill kunna utföra mer komplex validering i din presentationsmodell.

Presentationsmodells-API

Basklasser för presentationsmodeller. Ligger i Client Modules/Core Cluster/Core Util

DateSpan och Datum-API

API för att jobba med datum och tidsperioder. Ligger i ProSang Libraries/ProSang Utils. Intressanta klasser är DateUtil, DateFormatUtil och DateSpan.

EntityDisplayer-API

SPI för att utan att känna till vem som visar en entitet be någon annan att visa den. Gör det möjligt att registrera en EntityDisplayer för en viss entitetsklass för andra att använda. Ligger i Client Modules/Core Cluster/Core Util och paketet se.databyran.prosang.client.core.util.spi